# Mikrokontroléry a embedded systémy

## Architektury a jádro ARM Cortex-M, zásady návrhu firmware

Ing. Aleš POVALAČ, Ph.D.
povalac@vut.cz

Ústav radioelektroniky
Vysoké učení technické v Brně, FEKT

Školení pro SPŠ Třebíč

březen 2023

# Zásady návrhu embedded systémů

# Postup návrhu programu

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

- sepsat **předem**, co všechno má zařízení umět, přiřadit priority, navrhnout strukturu firmwaru
- méně důležité věci později (tzv. inkrementální vývoj)
- **postup tvorby** firmwaru pro nový hardware
  1. rozblikání LEDky
  2. rozchození „operačního systému“ – řízení úloh
  3. příprava základních ovladačů periferií a jejich testování
  4. kostra hlavního programu pro nejdůležitější funkce
  5. ladění a doplňování dalších ovladačů a funkcí

# Obsah *.c a *.h souborů

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

- zdrojové soubory *.c

1. dokumentační část – popis
2. systémové **#include <...>**
3. uživatelské **#include "..."**
4. (globální proměnné)
5. lokální proměnné – **static**
6. lokální funkce – **static**
7. globální funkce

- hlavičkové soubory *.h

1. dokumentační část – popis
2. ochrana před vícenásobným načtením

```
#ifndef _SOUBOR_H
#define _SOUBOR_H
... hlavičkový soubor ...
#endif
```

3. konstanty **#define**
4. makra **#define**
5. globální typy **typedef**
6. (globální proměnné)
7. prototypy globálních funkcí – **extern**

# Datové typy



- problematická portabilita (int) i čitelnost
- nejednoznačnost signed vs. unsigned
- raději jednoznačné číselné typy, dle standardu MISRA
- **`#include <stdint.h>`**

| | | |
|---|---|---|
| **`int8_t`** | 8-bit signed integer | `signed char` |
| **`int16_t`** | 16-bit signed integer | `signed int` |
| **`int32_t`** | 32-bit signed integer | `signed long` |
| **`uint8_t`** | 8-bit unsigned integer | `unsigned char` |
| **`uint16_t`** | 16-bit unsigned integer | `unsigned int` |
| **`uint32_t`** | 32-bit unsigned integer | `unsigned long` |

# Nezbytnosti

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

- standard psaní firmwaru – „štábní kultura“
  - hlavně v celém projektu stejně
  - pečlivě odsazovat
  - složitý kód rozepisovat na jednodušší a komentovat
  - vhodná volba názvů funkcí a proměnných
  - čím složitější funkce, tím víc komentářů
- tiché pracovní prostředí
  - po vyrušení trvá návrat k původnímu toku myšlenek cca 15 minut
- identifikace špatného kódu
  - 80% chyb v 20% kódu (Boehm), ve 2% kódu (Weinberg)
  - 57% chyb v 7% kódu (IBM)
  - identifikovat, bez milosti **smazat** a začít znovu

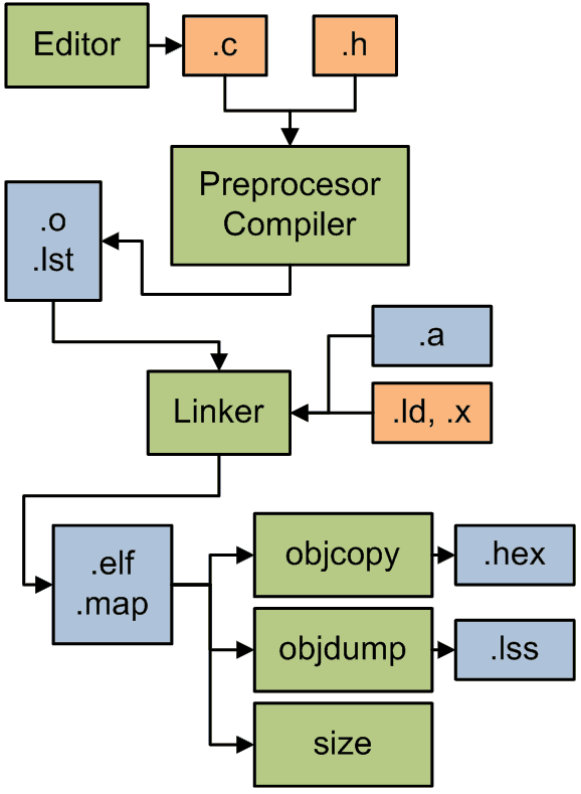
# Dělení programů



- zapouzdřování kódu
  - zcela nezbytné
  - skrytí interního kódu a proměnných uvnitř modulu, třída *static*
- přístup k proměnným pouze přes funkce, **nepoužívat** globální proměnné
  - změna „kdekoliv“ a „kdykoliv“, šílené ladění, problémy s reentrancí i atomičností
  - když už, tak dokumentované, zdůvodněné, vždy v jediném souboru projektu
- přiměřený rozsah funkce do 2 „obrazovek“
- u modulu do 10 „obrazovek“ ~ 500 LoC ~ 20 kB
- vhodná volba MCU
  - u prototypů a malosériové výroby **není cena HW důležitá**
  - nepřekračovat využití 75% prostředků MCU
  - 70% prostředků OK, 90% zdvojnásobí čas vývoje, 95% ztrojnásobí

# Oddělený překlad (STM32)

- průběh kompilace v GCC




- linker skript (*.ld; *.x)
- definice typů paměti a počátku

```
MEMORY
{
  rom (rx):  ORIGIN = 0x08000000,
             LENGTH = 64K
  ram (rwx): ORIGIN = 0x20000000,
             LENGTH = 20K
}
```

- definice sekcí

```
SECTIONS
{
  .vectors:    {...} >rom
  .text:       {...} >rom
  .rodata:     {...} >rom
  .data:       {...} >ram
  .bss:        {...} >ram
}
```

# Přerušení



- **krááááátké !!!** ... nejlíp tak 10-20 LoC
- nevolat jiné funkce, nepoužívat cykly, nikdy nepoužívat delay() v ISR
- atomičnost operace: přístup k 16-bit registru na 8-bit jádru (např. AVR)
- reentrance: atomičnost + statické proměnné
- ke každému ISR nutné znát
  - typická doba zpracování
  - typická četnost volání
- C nebo assembler?
  - u C není možné odhadnout dobu zpracování, nutné změřit
  - u ASM více chyb, méně čitelný kód, nutnost úschovy registrů

# Kritické – atomické sekce (8bit)



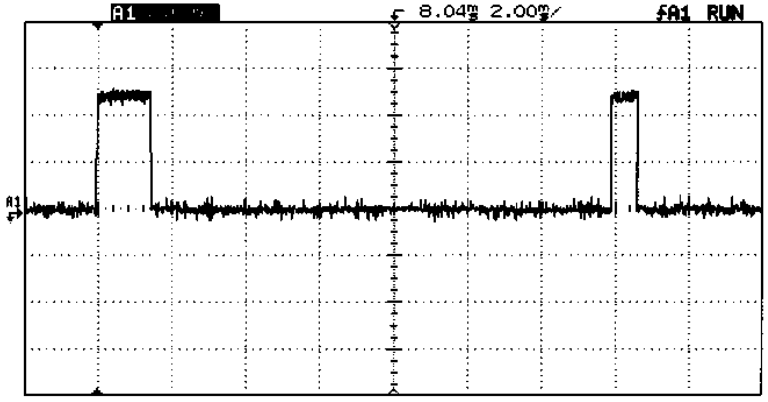
- problém u **8-bit MCU** => netýká se ARM Cortex-M
- sekce kódu, které nesmí být přerušeny
- atomický přístup k 16bitové proměnné – volatile nestačí!
- příklad problému:

  `do { ... } while (ctr != 0);`
  - hodnota klesne na 0x0100, nižní byte nulový
  - interrupt během testu lo/hi, dekrementace na 0x00FF, vyšší byte nulový

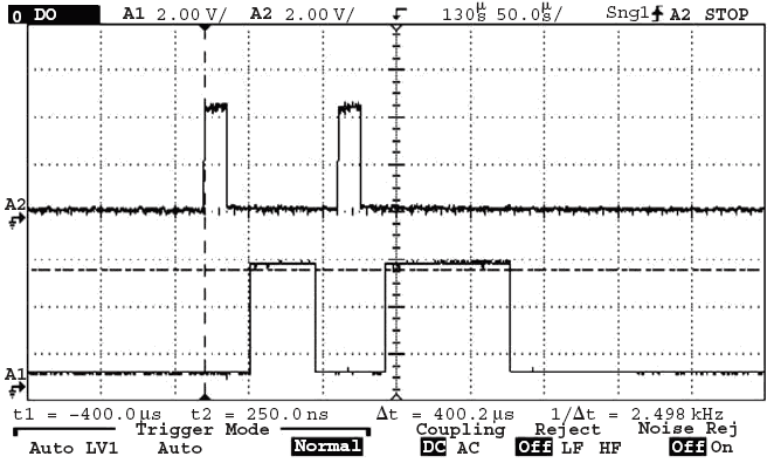
```
// příklad řešení pro AVR
volatile static uint16_t ctr = 0x0200;

ISR(TIMER1_OVF_vect) { ctr--; }

int main(void)
{
   uint16_t ctr_copy;

   ...
   sei();
   do {
     ...
     cli();
     ctr_copy = ctr;
     sei();
   } while (ctr_copy != 0);
   ...
}
```

# Ladění osciloskopem

- jak ladit ISR? – těžko, nejlépe nijak
- osciloskop na volném pinu




```
void SysTick_Handler(void)
{
  GPIOB->BSRR = (1<<0);
  // kod obsluhy preruseni
  GPIOB->BRR = (1<<0);
}
```




horní signál: IRQ linka
dolní signál: reakce a doba provádění ISR

zdroj: J. Ganssle, The Art of Designing Embedded Systems

# Standard firmwaru



- zdrojový kód musí nejen **fungovat**, ale i **popisovat jak funguje** budoucím programátorům (a budoucím nám)
- popisné názvy proměnných a funkcí v angličtině
- komentáře **anglicky** nebo česky bez diakritiky
- **nepoužívat "bulharské konstanty"** v kódu
- soubory projektu
  - nezasahovat do přejatých knihoven (výjimky dokumentovat)
  - dělení do modulů podle funkce
  - využívat **zapouzdření** (encapsulation), tj. mimo modul zpřístupnit jen vybraná volání; ostatní vnitřní funkce/proměnné privátní
  - přiměřeně **krátké** (do 1000 LoC)
- adresářová struktura
  - nepoužívat mezery, diakritiku, síťové disky
  - složka projekty, resp. projects v kořenu C: nebo D:

# Styl psaní kódu

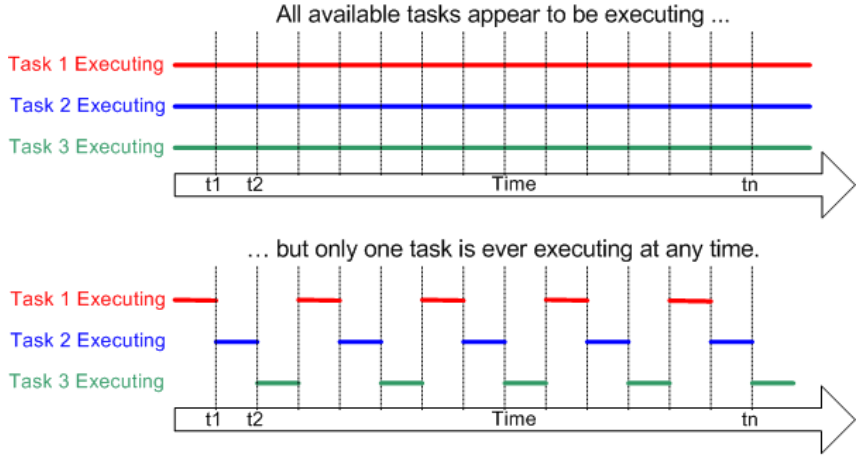


- odsazování pomocí dvou nebo **čtyř** mezer, ne tabulátor
- limit délky řádku (cca 120 znaků pro běžné LCD)
- ve výrazech závorky, nespoléhat na priority kompilátoru
- nikdy **nezanořovat if** do více než třetí úrovně
  - nahradit voláním funkce, výrazem switch
- při zanoření if-else do dalšího if výrazu vždy použít blok { }

```
if (cond1) {
    if (cond2) {
        codeA;
    } else {
        codeB;
    }
}
```

# Principy RTOS

- několik úloh, které musí běžet "současně"
- kooperativní a preemptivní; nutnost na malých MCU?
- dva stavy: čekání na I/O, využití CPU
- nejjednodušší kooperativní RTOS: "supersmyčka"
  - jednotlivé úlohy ve formě funkcí, volané v nekonečné smyčce funkce main() – z principu kooperativní




```
init_hw();
while (1) {
  ADC_Task();
  Status_Task();
  UI_Task();
}
```

zdroj: FreeRTOS, Multitasking Basics

# Stavový automat jako kooperativní RTOS



- použití neblokujícího časovače (300 ms zap., 200 ms vyp.)

```
void blikatko(void)
{
  static bool blik;
  static uint32_t DelayCnt = 0;

  if (blik) {
    if (HAL_GetTick() > DelayCnt + 300) {
       GPIOD->BRR = (1<<0);
       DelayCnt = HAL_GetTick();
       blik = false;
    }
  } else {
    if (HAL_GetTick() > DelayCnt + 200) {
       GPIOD->BSRR = (1<<0);
       DelayCnt = HAL_GetTick();
       blik = true;
    }
  }
}
```

volán z funkce main()

```
while (1) {
  blikatko();
  // dalsi funkce
  // ktere neblokuji
}
```